

# LU Factorization of Sparse, Unsymmetric Jacobian Matrices on Multicomputers: *Experience, Strategies, Performance*

STUDENT PAPER

Anthony Skjellum   Alvin P. Leung

*Advisor:* Manfred Morari

California Institute of Technology  
Chemical Engineering; mail code 210-41  
Pasadena, California 91125  
e-mail: [tony@perseus.ccsf.caltech.edu](mailto:tony@perseus.ccsf.caltech.edu)

## Abstract

Efficient sparse linear algebra *cannot* be achieved as a straightforward extension of the dense case, even for concurrent implementations. This paper details a new, general-purpose unsymmetric sparse LU factorization code built on the philosophy of Harwell's MA28, with variations. We apply this code in the framework of Jacobian-matrix factorizations, arising from Newton iterations in the solution of nonlinear systems of equations. Serious attention has been paid to the data-structure requirements, complexity issues and communication features of the algorithm. Key results include reduced communication pivoting for both the "analyze" A-mode and repeated B-mode factorizations, and effective general-purpose data distributions useful incrementally to trade-off process-column load balance in factorization against triangular solve performance. Future planned efforts are cited in conclusion.

## Introduction

The topic of this paper is the implementation and concurrent performance of sparse, unsymmetric LU factorization for medium-grain multicomputers. Our target hardware is distributed-memory, message-passing concurrent computers such as the Symult s2010 and Intel iPSC/2 systems. For both of these systems, efficient cut-through *wormhole* routing technology provides pair-wise communication performance essentially independent of the spatial location of the computers in the ensemble [2]. The Symult s2010 is a two-dimensional, mesh-connected concurrent computer; all examples in this paper were run on this variety of hardware. Message-passing performance, portability and

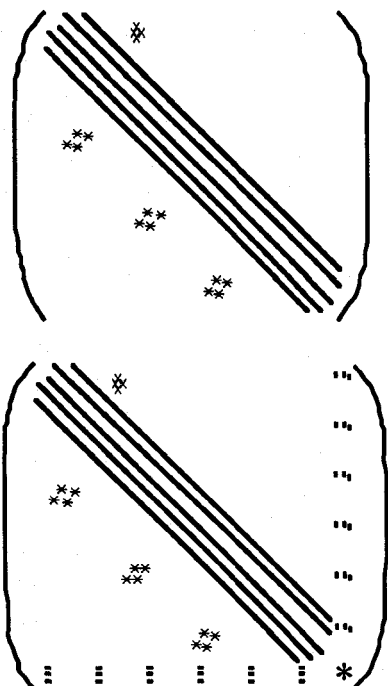
related issues relevant to this work are detailed in [7].

Questions of linear-algebra performance are pervasive throughout scientific and engineering computation. The need for high-quality, high-performance linear algebra algorithms (and libraries) for multicomputer systems therefore requires no attempt at justification. The motivation for the work described here has a specific origin, however. Our main higher-level research goal is the concurrent dynamic simulation of systems modelled by ordinary differential and algebraic equations; specifically, dynamic flowsheet simulation of chemical plants (*e.g.*, coupled distillation columns) [8]. Efficient sequential integration algorithms solve staticized nonlinear equations at each time point via modified Newton iteration (*cf.*, [3], Chapter 5). Consequently, a sequence of structurally identical linear systems must be solved; the matrices are finite-difference approximations to Jacobians of the staticized system of ordinary differential-algebraic equations. These Jacobians are large, sparse and unsymmetric for our application area. In general, they possess both band and significant off-band structure. Generic structures are depicted in Figure 0. This work should also bear relevance to electric power network/grid dynamic simulation where sparse, unsymmetric Jacobians also arise, and also elsewhere.

## Design Overview

We solve the problem  $Ax = b$  where  $A$  is large, and includes many zero entries. We assume that  $A$  is unsymmetric both in sparsity pattern and in numerical values. In general, the matrix  $A$  will be computed in a distributed fashion, so we will inherit a distribution of the coefficients of  $A$  (*cf.*, Figures 2., 3.). Follow-

Figure 0. Example Jacobian Matrix Structures.



In chemical-engineering process flowsheets, Jacobians with main band structure, and lower-triangular structure (feedforwards), upper-triangular structure (feedbacks), and borders (global or artificially restructured feedforwards and/or feedbacks) are common.

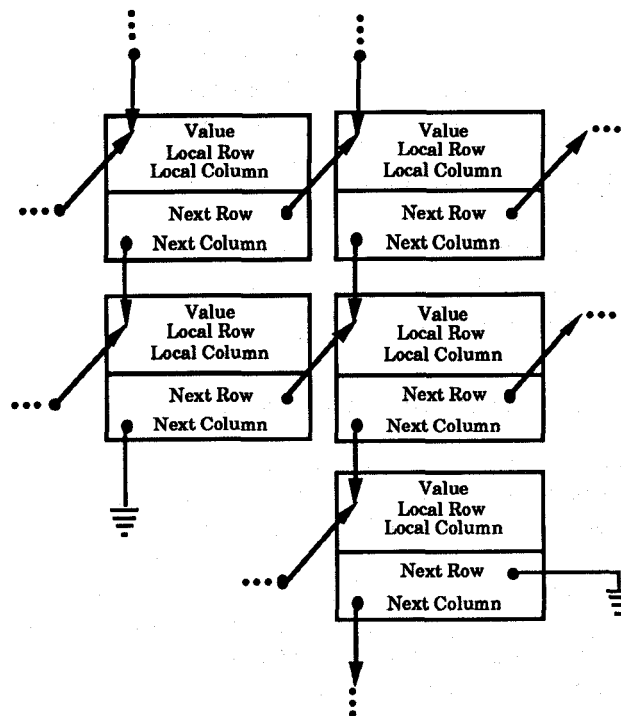
ing the style of Harwell's MA28 code for unsymmetric sparse matrices, we use a two-phase approach to this solution. There is a first LU factorization called A-mode or "analyze," which builds data structures dynamically, and uses a user-defined pivoting function. The repeated B-mode factorization uses the existing data structures statically to factor a new, similarly structured matrix, with the previous pivoting pattern. B-mode monitors stability with a simple growth factor estimate. In practice, A-mode is repeated whenever instability is detected. The two key contributions of this sparse concurrent solver are: reduced communication pivoting, and new data distributions for better overall performance.

Following Van de Velde [11], we consider the LU factorization of a real matrix  $A$ ,  $A \in \mathbb{R}^{N \times N}$ . It is well known (e.g., [6], pp. 117-118), that for any such matrix  $A$ , an LU factorization of the form

$$P_R A P_C^T = \hat{L} \hat{U}$$

exists, where  $P_R, P_C$  are square, (orthogonal) permutation matrices, and  $\hat{L}, \hat{U}$  are the unit lower-triangular,

Figure 1. Linked-list Entry Structure of Sparse Matrix.



A single entry consists of a double-precision value (8 bytes), the local row ( $i$ ) and column ( $j$ ) index (2 bytes each), a "Next Column Pointer" indicating the next current column entry (fixed  $j$ ), and a "Next Row Pointer" indicating the next current row entry (fixed  $i$ ), at 4 bytes each. Total: 24 bytes per entry.

and upper-triangular factors, respectively. Whereas the pivot sequence is stored (two  $N$ -length integer vectors), the permutation matrices are not stored or computed with explicitly. Rearranging, based on the orthogonality of the permutation matrices,  $A = P_R^T \hat{L} \hat{U} P_C$ . We factor  $A$  with implicit pivoting (no rows or columns are exchanged explicitly as a result of pivoting). Therefore, we do not store  $\hat{L}, \hat{U}$  directly, but instead:  $L = P_R^T \hat{L} P_C$ ,  $U = P_R^T \hat{U} P_C$ . Consequently,  $\hat{L} = P_R L P_C^T$ ,  $\hat{U} = P_R U P_C^T$ , and  $A = L (P_C^T P_R) U$ . The "unravelling" of the permutation matrices is accomplished readily (without implication of additional interprocess communication) during the triangular solves.

For the sparse case, performance is more difficult to quantify than for the dense case, but, for example, banded matrices with bandwidth  $\beta$  can be factored with  $O(\beta^2 N)$  work; we expect sub-cubic complexity in  $N$  for reasonably sparse matrices, and strive for sub-quadratic complexity, for very sparse matrices. The

triangular solves can be accomplished in work proportional to the number of entries in the respective triangular matrix  $L$  or  $U$ . The pivoting strategy is treated as a parameter of the algorithm and is not pre-determined. We can consequently treat the pivoting function as an application-dependent function, and sometimes tailor it to special problem structures (cf., Section 7 of [9]) for higher performance. As for all sparse solvers, we also seek sub-quadratic memory requirements in  $N$ , attained by storing matrix entries in linked-list fashion, as illustrated in Figure 1.

For further discussion of LU factorizations and sparse matrices, see [6,4].

## Reduced-Communication Pivoting

At each stage of the concurrent LU factorization, the pivot element is chosen by the user-defined pivot function. Then, the pivot row (new row of  $U$ ) must be broadcast, and pivot column (new column of  $L$ ) must be computed and broadcast on the logical process grid (cf., Figure 2.), vertically and horizontally, respectively. Note that these are interchangeable operations. We use this degree-of-freedom to reduce the communication complexity of particular pivoting strategies, while impacting the effort of the LU factorization itself negligibly.

We define two "correctness modes" of pivoting functions. In the first correctness mode "first row fanout," the exit conditions for the pivot function are: all processes must know  $\hat{p}$  (the pivot process row), the pivot process row must know  $\hat{q}$  (the pivot process column) as well as  $\hat{i}$ , the  $\hat{p}$ -local matrix row of the pivot, and the pivot process must know in addition the pivot value and  $\hat{q}$ -local matrix column  $\hat{j}$  of the pivot. Partial column pivoting and preset pivoting can be setup to satisfy these correctness conditions as follows. For partial column pivoting, the  $k$ th row is eliminated at the  $k$ th step of the factorization. From this fact, each process can derive the process row  $\hat{p}$  and  $\hat{p}$ -local matrix row  $\hat{i}$  using the row data distribution function. Having identified themselves, the pivot-row processes can look for the largest element in local matrix row  $\hat{i}$  and choose the pivot element globally among themselves via a *combine*. At completion this places  $\hat{q}$ ,  $\hat{j}$  and the pivot value in the entire pivot process row. This completes the requirements for the "first row fanout" correctness mode. For preset pivoting, the  $k$ th elimination row and column are both stored as  $\hat{p}, \hat{i}, \hat{q}, \hat{j}$ , and each process knows these values *without communication*.<sup>1</sup> Furthermore, the pivot process looks up the pivot value.

<sup>1</sup>Memory unscalabilities can be removed very cheaply; see [8].

Hence, preset pivoting satisfies the requirements of this correctness mode also.

For "first row fanout," the universal knowledge of  $\hat{p}$  and knowledge of the pivot matrix row  $\hat{i}$  by the pivot process row, allows the vertical broadcast of this row (new row of  $U$ ). In addition, we broadcast  $\hat{q}$ ,  $\hat{j}$  and the pivot value simultaneously. This extends the correct value of  $\hat{q}$  to all processes, as well as  $\hat{j}$  and the pivot value to the pivot process column. Hence, the multiplier ( $L$ ) column may be correctly computed and broadcast. Along with the multiplier column broadcast, we include the pivot value. After this broadcast, all processes have the correct indices  $\hat{p}, \hat{i}, \hat{q}, \hat{j}$  and the pivot value. This provides all that's required to complete the current elimination step.

For the second correctness mode "first column fanout," the exit conditions for the pivot function are: all processes must know  $\hat{q}$ , the entire pivot process column must know  $\hat{j}$ , the pivot value, and  $\hat{p}$ . The pivot process in addition knows  $\hat{i}$ . Partial row pivoting can be setup to satisfy these correctness conditions. The arguments are analogous to partial column pivoting and are given in [8].

For "first column fanout," the entire pivot process column knows the pivot value, and local column of the pivot. Hence, the multiplier column may be computed by dividing the pivot matrix column by the pivot value. This column of  $L$  may then be broadcast horizontally, including the pivot value,  $\hat{p}$  and  $\hat{i}$  as additional information. After this step, the entire ensemble has the correct pivot value, and  $\hat{p}$ ; in addition, the pivot process row has the correct  $\hat{i}$ . Hence, the pivot matrix row may be identified and broadcast. This second broadcast completes the needed information in each process for effecting the  $k$ th elimination step.

Hence, when using partial row or partial column pivoting, only local combines of the pivot process column (respectively row) are needed. The other processes don't participate in the combine, as they must without this methodology. Preset pivoting implies no pivoting communication, except very occasionally (e.g., 1 in 5000 times) as noted in [8] to remove memory unscalabilities. This pivoting approach is a direct savings, gained at a negligible additional broadcast overhead. See also [8].

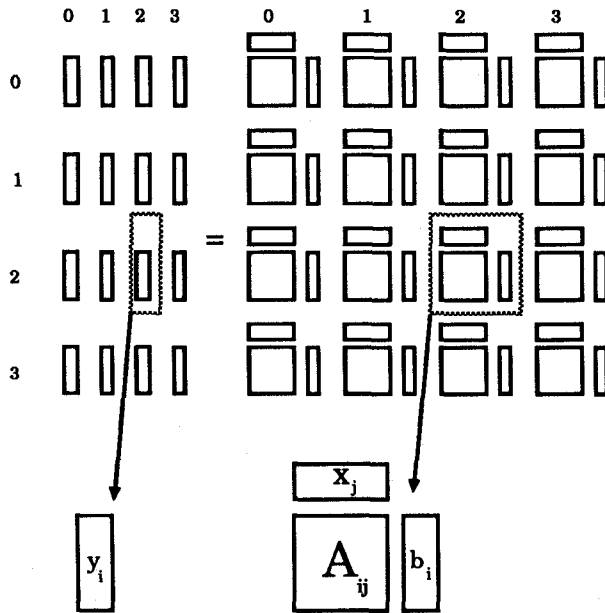
## New Data Distributions

We introduce new closed-form  $O(1)$ -time,  $O(1)$ -memory data distributions useful for sparse matrix factorizations and the problems that generate such matrices. We quantify evaluation costs in Table 0. Every concurrent data structure is associated with a logi-

Table 0. Data-Distribution Function and Inverse Costs		
Distribution:	$\mu(I, P, M)$	$\mu^{-1}(p, i, P, M)$
One-Parameter ( $\zeta$ )	$5.5554 \times 10^1 \pm 5 \times 10^{-3}$	$4.0024 \times 10^1 \pm 7 \times 10^{-3}$
Two-Parameter ( $\xi$ )	$6.1710 \times 10^1 \pm 1 \times 10^{-2}$	$4.2370 \times 10^1 \pm 8 \times 10^{-3}$
Block-Linear ( $\lambda$ )	$5.4254 \times 10^1 \pm 7 \times 10^{-3}$	$3.5404 \times 10^1 \pm 5 \times 10^{-3}$

For the data distributions and inverses described here, evaluation time in  $\mu s$  is quoted for the Symult s2010 multicomputer. Cardinality function calls are inexpensive, and fall within lower-order work anyway – their timing is hence omitted. The cheapest distribution function (scatter) costs  $\approx 15\mu s$  by way of comparison.

Figure 2. Process Grid Data Distribution of  $Ax = b$ .



Representation of a concurrent matrix, and distributed-replicated concurrent vectors on a 4x4 logical process grid. The solution of  $Ax = b$  first appears in  $x$ , a column-distributed vector, and then is normally “transposed” via a global combine to the row-distributed vector  $y$ .

cal process grid at creation (cf., Figure 2. and [7,8]). Vectors are either row- or column-distributed within a two-dimensional process grid. Row-distributed vectors are *replicated* in each process column, and distributed in the process rows. Conversely, column-distributed vectors are replicated in each process row, and distributed in the process columns. Matrices are distributed both in rows and columns, so that a single process owns a subset of matrix rows and columns. This partitioning follows the ideas proposed by Fox *et al.* [5] and others. Within the process grid, coefficients

of vectors and matrices are distributed according to one of several data distributions. Data distributions are chosen to compromise between load-balancing requirements and constraints on where information can be calculated in the ensemble.

#### Definition 1 (Data-Distribution Function)

A data-distribution function  $\mu$  maps three integers  $\mu(I, P, M) \mapsto (p, i)$  where  $I$ ,  $0 \leq I < M$ , is the global name of a coefficient,  $P$  is the number of processes among which all coefficients are to be partitioned, and  $M$  is the total number of coefficients. The pair  $(p, i)$  represents the process  $p$  ( $0 \leq p < P$ ) and local (process- $p$ ) name  $i$  of the coefficient ( $0 \leq i < \mu^{\dagger}(p, P, M)$ ). The inverse distribution function  $\mu^{-1}(p, i, P, M) \mapsto I$  transforms the local name  $i$  back to the global coefficient name  $I$ .

The formal requirements for a data distribution function are as follows. Let  $\mathcal{I}^P$  be the set of global coefficient names associated with process  $p$ ,  $0 \leq p < P$ , defined implicitly by a data distribution function  $\mu(\bullet, P, M)$ . The following set properties must hold:

$$\begin{aligned} \mathcal{I}^{p_1} \cap \mathcal{I}^{p_2} &= \emptyset, \forall p_1 \neq p_2, \quad 0 \leq p_1, p_2 < P \\ \bigcup_{p=0}^{P-1} \mathcal{I}^p &= \{0, \dots, M-1\} \equiv \mathcal{I}_M \end{aligned}$$

The cardinality of the set  $\mathcal{I}^p$ , is given by  $\mu^{\dagger}(p, P, M)$ .

The linear and scatter data-distribution functions are most often defined. We generalize these functions (by blocking and scattering parameters) to incorporate practically important degrees of freedom. These generalized distribution functions yield optimal static load balance as do the unmodified functions described in [11] for unit block size, but differ in coefficient placement. This distinction is technical, but necessary for efficient implementations.

### Definition 2 (Generalized Block-Linear)

The definitions for the generalized block-linear distribution function, inverse, and cardinality function are:

$$\begin{aligned}\lambda_B(I, P, M) &\mapsto (p, i), \\ p &\equiv P - 1 - \\ &\quad \max \left( \left\lfloor \frac{I_B^{\text{rev}}}{l+1} \right\rfloor, \left\lfloor \frac{I_B^{\text{rev}} - r}{l} \right\rfloor \right), \\ i &\equiv I - B(pl + \Theta^1(p - (P - r))),\end{aligned}$$

while

$$\begin{aligned}\lambda_B^{-1}(p, i, P, M) &\equiv i + B(pl + \Theta^1(p - (P - r))), \\ \lambda_B^{\dagger}(p, P, M) &\equiv B \left( \left\lfloor \frac{b+p}{P} \right\rfloor - \theta \right) + \\ &\quad (M \bmod B)\theta,\end{aligned}$$

where  $B$  denotes the coefficient block size,

$$\begin{aligned}b &= \begin{cases} \frac{M}{B} & \text{if } M \bmod B = 0 \\ \left\lfloor \frac{M}{B} \right\rfloor + 1 & \text{otherwise,} \end{cases} \\ I_B &= \left\lfloor \frac{I}{B} \right\rfloor, \quad I_B^{\text{rev}} = b - 1 - I_B, \\ l &= \left\lfloor \frac{b}{P} \right\rfloor, \quad r = b \bmod P, \\ \Theta^k(t) &\equiv \begin{cases} 0 & t \leq 0 \\ t^k & t > 0, k > 0 \\ 1 & t > 0, k = 0 \end{cases}, \\ \theta &= \left\lfloor \frac{p+1}{P} \right\rfloor \Theta^0(M \bmod B)\end{aligned}$$

and where  $b \geq P$ .

For  $B = 1$ , a load-balance-equivalent variant of the common linear data-distribution function is recovered. The general block-linear distribution function divides coefficients among the  $P$  processes  $p = 0, \dots, P - 1$  so that each  $\mathcal{I}^p$  is a set of coefficients with contiguous global names, while optimally load-balancing the  $b$  blocks among the  $P$  sets. Coefficient boundaries between processes are on multiples of  $B$ . The maximum possible coefficient imbalance between processes is  $B$ . If  $B \bmod P \neq 0$ , the last block in process  $P - 1$  will be foreshortened.

### Definition 3 (Parametric Functions)

To allow greater freedom in the distribution of coefficients among processes, we define a new, two-parameter distribution function family,  $\xi$ . The  $B$  blocking parameter (just introduced in the block-linear

function) is mainly suited to the clustering of coefficients that must not be separated by an interprocess boundary (again, see [8] for a definition of general block-scatter,  $\sigma$ ). Increasing  $B$  worsens the static load balance. Adding a second scaling parameter  $S$  (of no impact on the static load balance) allows the distribution to scatter coefficients to a greater or lesser degree, directly as a function of this one parameter. The two-parameter distribution function, inverse and cardinality function are defined below. The one-parameter distribution function family,  $\zeta$ , occurs as the special case  $B = 1$ , also as noted below:

$$\xi_{B,S}(I, P, M) \mapsto (p, i) \equiv \begin{cases} (p_0, i_0) & \Lambda_0 \geq l_S \\ (p_1, i_1) & \Lambda_0 < l_S \end{cases}$$

where

$$\begin{aligned}l_S &\equiv \left\lfloor \frac{l}{S} \right\rfloor, \quad \Lambda_0 \equiv \left\lfloor \frac{i_0}{BS} \right\rfloor, \\ (p_0, i_0) &\leftarrow \lambda_B(I, P, M), \\ I_{BS} &= p_0 l_S + \Lambda_0, \\ p_1 &\equiv I_{BS} \bmod P, \\ i_1 &\equiv BS \left\lfloor \frac{I_{BS}}{P} \right\rfloor + (i_0 \bmod BS),\end{aligned}$$

with

$$\begin{aligned}\xi_{B,S}(I, P, M) &\equiv \xi_{1,S}(I, P, M), \\ \xi_{B,S}^{\dagger}(p, P, M) &\equiv \zeta_S^{\dagger}(p, P, M) \\ &\equiv \lambda_B^{\dagger}(p, P, M),\end{aligned}$$

and where  $r, b$ , etc. are as defined above. The inverse distribution function  $\xi^{-1}$  is defined as follows:

$$\begin{aligned}\xi_{B,S}^{-1}(p, i, P, M) &\mapsto I = \lambda_B^{-1}(p^*, i^*, P, M), \\ (p^*, i^*) &\equiv \begin{cases} (p, i) & \Lambda \geq l_S \\ (p_2, i_2) & \Lambda < l_S \end{cases}, \\ \Lambda &\equiv \left\lfloor \frac{i}{BS} \right\rfloor, \quad I_{BS}^* = p + \Lambda P, \\ p_2 &\equiv \left\lfloor \frac{I_{BS}^*}{l_S} \right\rfloor, \\ i_2 &\equiv BS(I_{BS}^* \bmod l_S) + (i \bmod BS),\end{aligned}$$

with

$$\zeta_S^{-1}(p, i, P, M) \equiv \xi_{1,S}^{-1}(p, i, P, M).$$

For  $S = 1$ , a block-scatter distribution results, while for  $S \geq S_{\text{crit}} \equiv l_S + 1$ , the generalized block-linear distribution function is recovered. See also [8].

#### Definition 4 (Data Distributions)

Given a data-distribution function family  $(\mu, \mu^{-1}, \mu^{\dagger})$   $((\nu, \nu^{-1}, \nu^{\dagger}))$ , a process list of  $P$  ( $Q$ ),  $M$  ( $N$ ) as the number of coefficients, and a row (respectively, column) orientation, a row (column) data distribution  $\mathcal{G}^{\text{row}}$  ( $\mathcal{G}^{\text{col}}$ ) is defined as:

$$\mathcal{G}^{\text{row}} \equiv \{(\mu, \mu^{-1}, \mu^{\dagger}); P, M\},$$

respectively,

$$\mathcal{G}^{\text{col}} \equiv \{(\nu, \nu^{-1}, \nu^{\dagger}); Q, N\}.$$

A two-dimensional data distribution may be identified as consisting of a row and column distribution defined over a two-dimensional process grid of  $P \times Q$  processes, as  $\mathcal{G} \equiv (\mathcal{G}^{\text{row}}, \mathcal{G}^{\text{col}})$ .

Further discussion and detailed comparisons on data-distribution functions are offered in [8]. Figure 3. illustrates the effects of linear and scatter data-distribution functions on a small rectangular array of coefficients.

### Performance vs. Scattering

Consider a fixed logical process grid of  $R$  processes, with  $P \times Q = R$ . For the sake of argument, assume partial row pivoting during LU factorization for the retention of numerical stability. Then, for the LU factorization, it is well known that a scatter distribution is “good” for the matrix rows, and optimal were there no off-diagonal pivots chosen. Furthermore, the optimal column distribution is also scatter, because columns are chosen in order for partial row pivoting. Compatibly, a scatter distribution of matrix rows is also “good” for the triangular solves. However, for triangular solves, the best column distribution is linear, because this implies less intercolumn communication, as we detail below. In short, the optimal configurations conflict, and because explicit redistribution is expensive, a static compromise must be chosen. We address this need to compromise through the one-parameter distribution function  $\zeta$  described in the previous section, offering a variable degree of scattering via the  $S$ -parameter. To first order, changing  $S$  does not affect the cost of computing the Jacobian (assuming column-wise finite-difference computation), because each process column works independently.

It’s important to note that triangular solves derive no benefit from  $Q > 1$ . The standard column-oriented solve keep one process column active at any given time. For any column distribution, the updated right-hand-side vectors are retransmitted  $W$  times (process column-to-process column) during the triangular solve

– whenever the active process column changes. There are at least  $W_{\min} \equiv Q - 1$  such transmissions (linear distribution), and at most  $W_{\max} \equiv N - 1$  transmissions (scatter distribution). The complexity of this retransmission is  $O(WN/P)$ , representing quadratic work in  $N$  for  $W \sim N$ .

Calculation complexity for a sparse triangular solve is proportional to the number of elements in the triangular matrix, with a low leading coefficient. Often, there are  $O(N^{1.x})$  with  $x < 1$  elements in the triangular matrices, including fill. This operation is then  $O(N^{1.x}/P)$ , which is less than quadratic in  $N$ . Consequently, for large  $W$ , the retransmission step is likely of greater cost than the original calculation. This retransmission effect constrains the amount of scattering and size of  $Q$  in order to have any chance of concurrent speedup in the triangular solves.

Using the one-parameter distribution with  $S \geq 1$  implies that  $W \approx N/S$ , so that the retransmission complexity is  $O(N^2/SP)$ . Consequently, we can bound the amount of retransmission work by picking  $S$  sufficiently large. Clearly,  $S = S_{\text{crit}}$  is a hard upper bound, because we reach the linear distribution limit at that value of the parameter. We suggest picking  $S \approx 10$  as a first guess, and  $S \sim \sqrt{N}$ , more optimistically. The former choice basically reduces retransmission effort by an order of magnitude. Both examples in the following section illustrate the effectiveness of choosing  $S$  by these heuristics.

The two-parameter  $\xi$  distribution can be used on the matrix rows to tradeoff load balance in the factorizations and triangular solves against the amount of (communication) effort needed to compute the Jacobian. In particular, a greater degree of scattering can dramatically increase the time required for a Jacobian computation (depending heavily on the underlying equation structure and problem), but importantly reduce load imbalance during the linear algebra steps. The communication overhead caused by multiple process rows suggests shifting toward smaller  $P$  and larger  $Q$  (a squatter grid), in which case greater concurrency is attained in the Jacobian computation, and the additional communication previously induced is then somewhat mitigated. The one-parameter distribution used on the matrix columns then proves effective in controlling the cost of the triangular solves by choosing the minimally allowable amount of column scattering.

Let’s make explicit the performance objectives we consider when tuning  $S$ , and, more generally, when tuning the grid shape  $P \times Q = R$ . In the modified Newton iteration, for instance, a Jacobian factorization is reused until convergence slows unacceptably. An “LU Factor-

Figure 3. Example of Process-Grid Data Distribution

$$\begin{pmatrix} A^{0,0} & A^{0,1} & A^{0,2} & A^{0,3} \\ A^{1,0} & A^{1,1} & A^{1,2} & A^{1,3} \\ A^{2,0} & A^{2,1} & A^{2,2} & A^{2,3} \\ A^{3,0} & A^{3,1} & A^{3,2} & A^{3,3} \end{pmatrix}_{\mathcal{G}} = \begin{pmatrix} a_{0,1} & a_{0,5} & a_{0,2} & a_{0,6} & a_{0,3} & a_{0,7} & a_{0,0} & a_{0,4} & a_{0,8} \\ a_{1,1} & a_{1,5} & a_{1,2} & a_{1,6} & a_{1,3} & a_{1,7} & a_{1,0} & a_{1,4} & a_{1,8} \\ a_{2,1} & a_{2,5} & a_{2,2} & a_{2,6} & a_{2,3} & a_{2,7} & a_{2,0} & a_{2,4} & a_{2,8} \\ a_{3,1} & a_{3,5} & a_{3,2} & a_{3,6} & a_{3,3} & a_{3,7} & a_{3,0} & a_{3,4} & a_{3,8} \\ a_{4,1} & a_{4,5} & a_{4,2} & a_{4,6} & a_{4,3} & a_{4,7} & a_{4,0} & a_{4,4} & a_{4,8} \\ a_{5,1} & a_{5,5} & a_{5,2} & a_{5,6} & a_{5,3} & a_{5,7} & a_{5,0} & a_{5,4} & a_{5,8} \\ a_{6,1} & a_{6,5} & a_{6,2} & a_{6,6} & a_{6,3} & a_{6,7} & a_{6,0} & a_{6,4} & a_{6,8} \\ a_{7,1} & a_{7,5} & a_{7,2} & a_{7,6} & a_{7,3} & a_{7,7} & a_{7,0} & a_{7,4} & a_{7,8} \\ a_{8,1} & a_{8,5} & a_{8,2} & a_{8,6} & a_{8,3} & a_{8,7} & a_{8,0} & a_{8,4} & a_{8,8} \\ a_{9,1} & a_{9,5} & a_{9,2} & a_{9,6} & a_{9,3} & a_{9,7} & a_{9,0} & a_{9,4} & a_{9,8} \\ a_{10,1} & a_{10,5} & a_{10,2} & a_{10,6} & a_{10,3} & a_{10,7} & a_{10,0} & a_{10,4} & a_{10,8} \end{pmatrix}$$

An  $11 \times 9$  array with block-linear rows ( $B = 2$ ) and scattered columns on a  $4 \times 4$  logical process grid. Local arrays are denoted at left by  $A^{p,q}$  where  $(p,q)$  is the grid position of the process on  $\mathcal{G} \equiv (\{(\lambda_2, \lambda_2^{-1}, \lambda_2^4); P = 4, M = 11\}, \{(\sigma_1, \sigma_1^{-1}, \sigma_1^4); Q = 4, N = 9\})$ . Subscripts (i.e.,  $a_{I,J}$ ) are the global  $(I, J)$  indices.

ization + Backsolve” step is followed by  $\eta$  “Forward + Backsolves,” with  $\eta \sim O(1)$  typically (and varying dynamically throughout the calculation). Assuming an averaged  $\eta$ , say  $\eta^*$  (perhaps as large as five [3]), then our first-level performance goal is a heuristic minimization of

$$T_{LU} + (\eta^* + 1)T_{Back} + \eta^*T_{Forward}$$

over  $S$  for fixed  $P, Q$ .  $\eta^* > 1$  more heavily weights the reduction of triangular solve costs *vs.* B-mode factorization than we might at first have assumed, placing a greater potential gain on the one-parameter distribution for higher overall performance. We generally want heuristically to optimize

$$T_{Jac} + T_{LU} + (\eta^* + 1)T_{Back} + \eta^*T_{Forward}$$

over  $S, P, Q, R$ . Then, the possibility of fine-tuning row and column distributions is important, as is the use of non-power-of-two grid shapes.

## Performance

### Order 13040 Example

We consider an order 13040 banded matrix with a bandwidth of 326 under partial row pivoting. For this example, we have compiled timing results for a  $16 \times 12$  process grid with random matrices (entries have range 0-10,000) using different values of  $S$  on the column distribution (see Table 1). We indicate timing for A-mode, B-mode, Backsolves and Forward- and Backsolves together (“Solve” heading). For this example,

$S = 30$  saves 76% of the triangular solve cost compared to  $S = 1$ , or approximately 186 seconds, roughly 6 seconds above the linear optimal. Simultaneously, we incur about 17 seconds additional cost in B-mode, while saving about 93 seconds in the Backsolve. Assuming  $\eta^* = 1$  ( $\eta^* = 0$ ), in the first above-mentioned objective function, we save about 262 (respectively, 76) seconds. Based on this example, and other experience, we conclude that this is a successful practical technique for improving overall sparse linear algebra performance. The following example further bolsters this conclusion.

### Order 2500 Example

Now, we turn to a timing example of an order 2500 sparse, random matrix. The matrix has a random diagonal, plus two-percent random fill of the off-diagonals; entries have a dynamic range of 0-10,000. Normally, data is averaged over random matrices for each grid shape (as noted), and over four repetitive runs for each random matrix. Partial row pivoting was used exclusively. Table 2. compiles timings for various grid shapes of row-scatter/column-scatter, and row-scatter / column- ( $S = 10$ ) distributions, for as few as nine nodes and as many as 128. Memory limitations set the lower bound on the number of nodes.

This example demonstrates that speedups are possible for this reasonably small sparse example with this general-purpose solver, and that the one-parameter distribution is key to achieving overall better performance even for this random, essentially unstructured example. Without the one-parameter distribution, triangular solver performance is poor, except in grid con-

Table 1. Order 13040 Band Matrix Performance					
Distribution:		(time in seconds)			
Row	Column	A-Mode	B-Mode	Back-Solve	Solve
Scatter	S=1	$1.140 \times 10^3$	$1.603 \times 10^2$	$1.196 \times 10^2$	$2.426 \times 10^2$
	S=10	$1.148 \times 10^3$	$1.696 \times 10^2$	$3.294 \times 10^1$	$6.912 \times 10^1$
	S=25	$1.091 \times 10^3$	$1.670 \times 10^2$	$2.713 \times 10^1$	$5.752 \times 10^1$
	S=30	$1.095 \times 10^3$	$1.769 \times 10^2$	$2.653 \times 10^1$	$5.631 \times 10^1$
	S=40	$1.116 \times 10^3$	$2.157 \times 10^2$	$2.573 \times 10^1$	$5.472 \times 10^1$
	S=50	$1.127 \times 10^3$	$2.157 \times 10^2$	$2.764 \times 10^1$	$5.743 \times 10^1$
	S=100	$1.279 \times 10^3$	$4.764 \times 10^2$	$2.520 \times 10^1$	$5.367 \times 10^1$
	Linear	$2.247 \times 10^3$	$1.161 \times 10^3$	$2.333 \times 10^1$	$4.993 \times 10^1$

The above timing data, for the 16x12 grid configuration with scattered rows, indicates the importance of the one-parameter distribution with  $S > 1$  for balancing factorization cost *vs.* triangular-solve cost. The random matrices, of order 13040, have an upper bandwidth of 164 and a lower bandwidth of 162. "Best" performance occurs in the range  $S \approx 25 \dots 40$ .

figurations where the factorization is itself degraded (e.g., 2x16). Furthermore, the choice of  $S = 10$  is universally reasonable for the  $Q > 1$  grid shapes illustrated here, so the distribution proves easy to tune for this type of matrix. We are able to maintain an almost constant speed for the triangular solves while increasing speed for both the A-mode and B-mode factorizations. We presume, based on experience, that triangular solve times are comparable to the sequential solution times – further study is needed in this area to see if and how performance can be improved. The consistent A-mode to B-mode ratio of approximately two is attributed primarily to reduced communication costs in B-mode, realized through the elimination of essentially all *combine* operations in B-mode.

While triangular-solve performance exemplifies sequentialism in the algorithm, it should be noted that we do achieve significant overall performance improvements between 9 nodes and 72 (12x6 grid) nodes, and that the repeatedly used B-mode factorization remains dominant compared to the triangular solves even for 128 nodes. Consequently, efforts aimed further to increase performance of the B-mode factorization (at the expense of additional A-mode work) are interesting to consider. For the factorizations, we also expect that we are achieving non-trivial speedups relative to one node, but we are unable to quantify this at present because of the memory limitations alluded to above.

## Future Work, Conclusions

There are several classes of future work to be considered. First, we need to take the A-mode "analyze" phase to its logical completion, by including pivot-order sorting of the L/U pointer structures to improve performance for systems that should demonstrate sub-quadratic sequential complexity. This will require minor modifications to B-mode (that already takes advantage of column-traversing elimination), to reduce testing for inactive rows as the elimination progresses. We already realize optimal computation work in the triangular solves, and we mitigate the effect of  $Q > 1$  quadratic communication work using the one-parameter distribution.

Second, we need to exploit "timelike" concurrency in linear algebra – multiple pivots. This has been addressed by Alaghband for shared-memory implementations of MA28 with  $O(N)$ -complexity heuristics [1]. These efforts must be reconsidered in the multicomputer setting and effective variations must be devised. This approach should prove an important source of additional speedup for many chemical engineering applications, because of the tendency towards extreme sparsity, with mainly band and/or block-diagonal structure.

Third, we could exploit new communication strategies and data redistribution. Within a process grid, we could incrementally redistribute L/U by utilizing the inherent broadcasts of  $L$  columns and  $U$  rows to improve load balance in the triangular solves at



Table 2. Order 2500 Matrix Performance

Shape	Distribution:		(time in seconds)				Avgs
	Row	Column	A-Mode	B-Mode	Back-Solve	Solve	
3x3	Scatter	Scatter	$3.567 \times 10^2$	$1.783 \times 10^2$	$1.997 \times 10^1$	$4.115 \times 10^1$	1
3x4		Scatter	$3.101 \times 10^2$	$1.303 \times 10^2$	$2.149 \times 10^1$	$4.452 \times 10^1$	1
4x3		Scatter	$2.778 \times 10^2$	$1.526 \times 10^2$	$1.728 \times 10^1$	$3.537 \times 10^1$	1
2x16		Scatter	$4.500 \times 10^2$	$3.350 \times 10^2$	$3.175 \times 10^0$	$1.101 \times 10^1$	1
12x1		Scatter	$2.636 \times 10^2$	$1.206 \times 10^2$	$4.0188 \times 10^0$	$8.340 \times 10^0$	3
16x1		Scatter	$2.085 \times 10^2$	$1.000 \times 10^2$	$4.856 \times 10^0$	$9.8744 \times 10^0$	3
8x2		Scatter $S = 10$	$2.013 \times 10^2$	$9.41 \times 10^1$	$1.127 \times 10^1$	$2.295 \times 10^1$	3
			$1.997 \times 10^2$	$9.63 \times 10^1$	$4.508 \times 10^0$	$9.399 \times 10^0$	3
4x4		Scatter $S = 10$	$2.371 \times 10^2$	$1.056 \times 10^2$	$1.225 \times 10^1$	$3.549 \times 10^1$	3
			$2.329 \times 10^2$	$1.104 \times 10^2$	$4.192 \times 10^0$	$9.406 \times 10^0$	3
4x6		Scatter $S = 10$	$1.456 \times 10^2$	$7.72 \times 10^1$	$1.723 \times 10^1$	$3.528 \times 10^1$	3
			$1.684 \times 10^2$	$8.85 \times 10^1$	$4.206 \times 10^0$	$9.303 \times 10^0$	3
12x2		Scatter $S = 10$	$1.490 \times 10^2$	$6.95 \times 10^1$	$9.08 \times 10^0$	$1.851 \times 10^1$	3
			$1.425 \times 10^2$	$6.54 \times 10^1$	$4.557 \times 10^0$	$9.439 \times 10^0$	3
12x3		Scatter $S = 10$	$1.0429 \times 10^2$	$5.39 \times 10^1$	$9.34 \times 10^0$	$1.898 \times 10^1$	3
			$1.0382 \times 10^2$	$5.42 \times 10^1$	$4.539 \times 10^0$	$9.390 \times 10^0$	3
8x8		Scatter $S = 10$	$1.154 \times 10^2$	$6.16 \times 10^1$	$1.1082 \times 10^1$	$2.2906 \times 10^1$	3
			$1.145 \times 10^2$	$6.64 \times 10^1$	$4.4600 \times 10^0$	$9.651 \times 10^0$	3
12x6		Scatter $S = 10$	$6.470 \times 10^1$	$3.527 \times 10^1$	$9.410 \times 10^0$	$1.9141 \times 10^1$	3
			$6.265 \times 10^1$	$3.417 \times 10^1$	$4.555 \times 10^0$	$9.495 \times 10^0$	3
16x8		Scatter $S = 10$	$7.046 \times 10^1$	$3.879 \times 10^1$	$8.9535 \times 10^0$	$1.8243 \times 10^1$	3
			$6.70 \times 10^1$	$3.854 \times 10^1$	$5.239 \times 10^0$	$1.0816 \times 10^1$	3

Performance as a function of grid shape and size, and  $S$ -parameter. “Best” performance is for the 12x6 grid with  $S = 10$ .

the expense of slightly more factorization computational overhead and significantly more memory overhead (nearly a factor of two). Memory overhead could be reduced at the expense of further communication if explicit pivoting were used concomitantly.

Fourth, we can develop adaptive broadcast algorithms that track the known load imbalance in the B-mode factorization, and shift greater communication emphasis to nodes with less computational work remaining. For example, the pivot column is naturally a “hot spot” because the multiplier column ( $L$  column) must be computed before broadcast to the awaiting process columns. Allowing the non-pivot columns to handle the majority of the communication could be beneficial, even though this implies additional overall communication. Similarly, we might likewise apply this to

the pivot row broadcast, and especially for the pivot process, because it must participate in two broadcast operations.

We could utilize two process grids. When rows (columns) of  $U$  ( $L$ ) are broadcast, extra broadcasts to a secondary process grid could reasonably be included. The secondary process grid could work on redistribution  $L/U$  to an efficient process grid shape and size for triangular solves while the factorization continues on the primary grid. This overlapping of communication and computation could also be used to reduce the cost of transposing the solution vector from column-distributed to row-distributed, which normally follows the triangular solves.

The sparse solver supports arbitrary user-defined pivoting strategies. We have considered but not fully

explored issues of fill-reduction *vs.* minimum time; in particular we have implemented a Markowitz-count fill-reduction strategy [4]. Study of the usefulness of partial column pivoting and other strategies is also needed. We will report on this in the future.

Reduced-communication pivoting and parametric distributions can be applied immediately to concurrent dense solvers with definite improvements in performance. While triangular solves remain lower-order work in the dense case, and may sensibly admit less tuning in  $S$ , the reduction of pivot communication is certain to improve performance. A new dense solver exploiting these ideas is under construction at present.

In closing, we suggest that the algorithms generating the sequences of sparse matrices must themselves be reconsidered in the concurrent setting. Changes that introduce multiple right-hand sides could help to amortize linear algebra cost over multiple time-like steps of the higher-level algorithm. Because of inevitable load imbalance, idle processor time is essentially free – algorithms that find ways to use this time by asking for more speculative (partial) solutions appear of merit toward higher performance.

### Acknowledgements

The authors acknowledge Prof. Manfred Morari, who supervised the work presented in this student paper. We wish to acknowledge the dense concurrent linear algebra library provided by Eric Van de Velde, as well as a prototype sparse concurrent linear algebra library, both of which were useful springboards for this work.

The first author acknowledges partial support under DOE grants DE-FG03-85ER25009 and DE-AC03-85ER40050. The second author (presently at the University of California, Santa Cruz) received support for his 1989 Caltech Summer Undergraduate Research Fellowship (SURF) under the same grants, and wishes to thank the Caltech SURF program for the opportunity to pursue the research discussed in part here.

The software implementation of this research was accomplished using machine resources made available by the Caltech Computer Science sub-Micron System Architectures Project and the Caltech Concurrent Supercomputer Facilities (CCSF).

### References

- [1] Alaghband, G., "Parallel pivoting combined with parallel reduction and fill-in control," *Parallel Computing* 11, 1989, pp. 201-221.

- [2] Athas W. C., and C. L. Seitz, "Multicomputers: Message-Passing Concurrent Computers," *IEEE Computer*, August 1988, pp. 9-24.
- [3] Brennan, K. E., S. L. Campbell, L. R. Petzold, *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*, Elsevier, 1989.
- [4] Duff, I.S., A. M. Erisman and J. K. Reid, *Direct Methods for Sparse Matrices*, Oxford University Press, 1986.
- [5] Fox, G., *et al.*, *Solving Problems on Concurrent Processors*, Volume 1, Prentice Hall, March 1988.
- [6] Golub, G. H., C. F. Van Loan, *Matrix Computations*, 2nd. Edition, John Hopkins University Press, 1989.
- [7] Skjellum, A., A. P. Leung, "Zipcode: A Portable Multicomputer Communication Library atop the Reactive Kernel," Proc. of DMCC5, Charleston, April 1990.
- [8] Skjellum, A., *Concurrent Dynamic Simulation: Multicomputer Algorithms Research Applied to Differential-Algebraic Process Systems in Chemical Engineering*, Ph.D. Dissertation, California Institute of Technology, Chemical Engineering, 1990.
- [9] Van de Velde, E. F., *A Concurrent Direct Solver for Sparse Unstructured Systems*, Caltech C<sup>3</sup>P Report #604, March 1988.
- [10] Van de Velde, E. F., *The Formal Correctness of an LU-Decomposition Algorithm*, Caltech C<sup>3</sup>P Report #625, June, 1988.
- [11] Van de Velde, E. F., *Experiments with Multicomputer LU-Decomposition*, Caltech / Rice Report CRPC-89-1. To appear in *Concurrency: Practice and Experience*.
- [12] Van de Velde, E. F., *Adaptive Data Distribution for Concurrent Continuation*, Caltech / Rice Center for Research in Parallel Computation Report CRPC-89-4. Submitted to *Num. Math.*